Relational Normalization Theory

Chapter 6

Limitations of E-R Designs

- Provides a set of guidelines, does not result in a unique database schema
- Does not provide a way of evaluating alternative schemas
- Normalization theory provides a mechanism for analyzing and refining the schema produced by an E-R design

Redundancy

- Dependencies between attributes cause redundancy
  - Ex. All addresses in the same town have the same zip code

Redundancy and Other Problems

- Set valued attributes in the E-R diagram result in multiple rows in corresponding table
- Example: Person (SSN, Name, Address, Hobbies)
  - A person entity with multiple hobbies yields multiple rows in table Person
    - Hence, the association between Name and Address for the same person is stored redundantly
  - SSN is key of entity set, but (SSN, Hobby) is key of corresponding relation
    - The relation Person can't describe people without hobbies

Example

ER Model

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>biking, hiking</td>
</tr>
</tbody>
</table>

Relational Model

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>biking</td>
</tr>
</tbody>
</table>

Anomalies

- Redundancy leads to anomalies:
  - Update anomaly: A change in Address must be made in several places
  - Deletion anomaly: Suppose a person gives up all hobbies. Do we:
    - Set Hobby attribute to null? No, since Hobby is part of key
    - Delete the entire row? No, since we lose other information in the row
  - Insertion anomaly: Hobby value must be supplied for any inserted row since Hobby is part of key
Decomposition

- **Solution**: use two relations to store Person information
  - Person1 (SSN, Name, Address)
  - Hobbies (SSN, Hobby)
- The decomposition is more general: people with hobbies can now be described
- No update anomalies:
  - Name and address stored once
  - A hobby can be separately supplied or deleted

Normalization Theory

- Result of E-R analysis need further refinement
- Appropriate decomposition can solve problems
- The underlying theory is referred to as normalization theory and is based on functional dependencies (and other kinds, like multivalued dependencies)

Functional Dependencies

- **Definition**: A functional dependency (FD) on a relation schema R is a constraint X → Y, where X and Y are subsets of attributes of R.
- **Definition**: An FD X → Y is satisfied in an instance r of R if for every pair of tuples, t and s: if t and s agree on all attributes in X then they must agree on all attributes in Y
  - Key constraint is a special kind of functional dependency: all attributes of relation occur on the right-hand side of the FD:
    - SSN → SSN, Name, Address

Functional Dependency - Example

- Consider a brokerage firm that allows multiple clients to share an account, but each account is managed from a single office and a client can have no more than one account in an office
  - HasAccount (AcctNum, ClientId, OfficeId)
    - keys are (ClientId, OfficeId), (AcctNum, ClientId)
  - Client, OfficeId → AcctNum
  - AcctNum → OfficeId
    - Thus, attribute values need not depend only on key values

Entailment, Closure, Equivalence

- **Definition**: If F is a set of FDs on schema R and f is another FD on R, then F entails f if every instance r of R that satisfies every FD in F also satisfies f
  - Ex: F = {A → B, B → C} and f is A → C
    - If Town → Zip and Zip → AreaCode then Town → AreaCode
- **Definition**: The closure of F, denoted F*, is the set of all FDs entailed by F
- **Definition**: F and G are equivalent if F entails G and G entails F
Entailment (cont’d)

- Satisfaction, entailment, and equivalence are semantic concepts – defined in terms of the actual relations in the “real world.”
  - They define what these notions are, not how to compute them.
- How to check if \( F \) entails \( f \) or if \( F \) and \( G \) are equivalent?
  - **Bad idea**: might be infinite number for infinite domains
  - **Solution**: find algorithmic, syntactic ways to compute these notions
    - **Important**: the syntactic solution must be “correct” with respect to the semantic definitions
    - Correctness has two aspects: soundness and completeness – see later.

Armstrong’s Axioms for FDs

- This is the syntactic way of computing/testing the various properties of FDs
  - **Reflexivity**: If \( Y \subseteq X \) then \( X \rightarrow Y \) (trivial FD)
    - Name, Address \( \rightarrow \) Name
  - **Augmentation**: If \( X \rightarrow Y \) then \( XZ \rightarrow YZ \)
    - If Town \( \rightarrow \) Zip then Town, Name \( \rightarrow \) Zip, Name
  - **Transitivity**: If \( X \rightarrow Y \) and \( Y \rightarrow Z \) then \( X \rightarrow Z \)

Soundness

- Axioms are sound: if an FD \( f: X \rightarrow Y \) can be derived from a set of FDs \( F \) using the axioms, then \( f \) holds in every relation that satisfies every FD in \( F \).
- Example: Given \( X \rightarrow Y \) and \( X \rightarrow Z \) then
  - \( X \rightarrow XY \) (Augmentation by \( X \))
  - \( YX \rightarrow YZ \) (Augmentation by \( Y \))
  - \( X \rightarrow YZ \) (Transitivity)
  - Thus, \( X \rightarrow YZ \) is satisfied in every relation where both \( X \rightarrow Y \) and \( X \rightarrow Z \) are satisfied
  - Therefore, we have derived the union rule for FDs: we can take the union of the RHSs of FDs that have the same LHS.

Completeness

- Axioms are complete: if \( F \) entails \( f \), then \( f \) can be derived from \( F \) using the axioms.
- A consequence of completeness is the following (naïve) algorithm to determining if \( F \) entails \( f \):
  - **Algorithm**: Use the axioms in all possible ways to generate \( F^+ \) (the set of possible FD’s is finite so this can be done) and see if \( f \) is in \( F^+ \).

Correctness

- The notions of soundness and completeness link the syntax (Armstrong’s axioms) with semantics (the definitions in terms of relational instances).
- This is a precise way of saying that the algorithm for entailment based on the axioms is “correct” with respect to the definitions.

Generating \( F^+ \)

\[
\begin{array}{c}
F \\
\hline
AB \rightarrow C \\
A \rightarrow D \\
D \rightarrow K \\
\end{array}
\]

union: \( AB \rightarrow BCD \)
trans: \( AB \rightarrow BCDE \)
decomp: \( AB \rightarrow CDE \)

Thus, \( AB \rightarrow BD, AB \rightarrow BCD, AB \rightarrow BCDE, \) and \( AB \rightarrow CDE \) are all elements of \( F^+ \).
Attribute Closure

- Calculating attribute closure leads to a more efficient way of checking entailment.
- The attribute closure of a set of attributes, X, with respect to a set of functional dependencies, F, (denoted $X^+_F$) is the set of all attributes, A, such that $X \rightarrow A$.
- $X^+_F$ is not necessarily the same as $X^+_{F_1}$ if $F_1 \neq F_2$.
- Attribute closure and entailment:
  - Algorithm: Given a set of FDs, F, then $X \rightarrow Y$ if and only if $X^+_F \supseteq Y$.

Example - Computing Attribute Closure

<table>
<thead>
<tr>
<th>F: $AB \rightarrow C$</th>
<th>$X$</th>
<th>$X^+_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow D$</td>
<td>$A$</td>
<td>${A, D, E}$</td>
</tr>
<tr>
<td>$D \rightarrow E$</td>
<td>$AB$</td>
<td>${A, B, C, D, E}$</td>
</tr>
<tr>
<td>$AC \rightarrow B$</td>
<td>$B$</td>
<td>${B}$</td>
</tr>
<tr>
<td></td>
<td>$D$</td>
<td>${D, E}$</td>
</tr>
</tbody>
</table>

Is $AB \rightarrow E$ entailed by $F$? Yes
Is $D \rightarrow C$ entailed by $F$? No
Result: $X^+_F$ allows us to determine FDs of the form $X \rightarrow Y$ entailed by $F$.

Computation of Attribute Closure $X^+_F$

```
closure := X;  // since $X \subseteq X^+_F$
repeat
  old := closure;
  if there is an FD $Z \rightarrow V$ in $F$ such that $Z \subseteq closure$ and $V \subseteq closure$
    then closure := closure $\cup$ $V$
until old = closure
If $T \subseteq closure$ then $X \rightarrow T$ is entailed by $F$.
```

Example: Computation of Attribute Closure

Problem: Compute the attribute closure of $AB$ with respect to the set of FDs:

- $AB \rightarrow C$
- $A \rightarrow D$
- $D \rightarrow E$
- $AC \rightarrow B$

Solution:
Initially closure = $\{AB\}$
Using (a) closure = $\{ABC\}$
Using (b) closure = $\{ABCD\}$
Using (c) closure = $\{ABCDE\}$

Normal Forms

- Each normal form is a set of conditions on a schema that guarantees certain properties (relating to redundancy and update anomalies).
- First normal form (1NF) is the same as the definition of relational model (relations = sets of tuples; each tuple = sequence of atomic values).
- Second normal form (2NF) – a research lab accident; has no practical or theoretical value – won’t discuss.
- The two commonly used normal forms are third normal form (3NF) and Boyce-Codd normal form (BCNF).

BCNF

- Definition: A relation schema $R$ is in BCNF if for every FD $X \rightarrow Y$ associated with $R$ either
  - $Y \subseteq X$ (i.e., the FD is trivial) or
  - $X$ is a superkey of $R$.
- Example: Person1($SSN, Name, Address$)
  - The only FD is $SSN \rightarrow Name, Address$
  - Since $SSN$ is a key, Person1 is in BCNF.
(non) BCNF Examples

- **Person** (SSN, Name, Address, Hobby)
  - The FD SSN → Name, Address does not satisfy requirements of BCNF
    - since the key is (SSN, Hobby)
- **HasAccount** (AcctNum, ClientId, OfficeId)
  - The FD AcctNum→OfficeId does not satisfy BCNF requirements
    - since keys are (ClientId, OfficeId) and (AcctNum, ClientId); not AcctNum.

Redundancy

- Suppose R has a FD A → B, and A is not a superkey. If an instance has 2 rows with same value in A, they must also have same value in B (⇒ redundancy, if the A-value repeats twice)

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>coins</td>
</tr>
</tbody>
</table>

- If A is a superkey, there cannot be two rows with same value of A
  - Hence, BCNF eliminates redundancy

Third Normal Form

- A relational schema R is in 3NF if for every FD X → Y associated with R either:
  - Y ⊆ X (i.e., the FD is trivial); or
  - X is a superkey of R; or
  - Every A ∈ Y is part of some key of R
- 3NF is weaker than BCNF (every schema that is in BCNF is also in 3NF)

3NF Example

- **HasAccount** (AcctNum, ClientId, OfficeId)
  - ClientId, OfficeId → AcctNum
    - OK since LHS contains a key
  - AcctNum → OfficeId
    - OK since RHS is part of a key
- **HasAccount** is in 3NF but it might still contain redundant information due to AcctNum → OfficeId (which is not allowed by BCNF)

3NF (Non) Example

- **Person** (SSN, Name, Address, Hobby)
  - (SSN, Hobby) is the only key.
  - SSN→Name violates 3NF conditions since Name is not part of a key and SSN is not a superkey

Decompositions

- **Goal**: Eliminate redundancy by decomposing a relation into several relations in a higher normal form
- Decomposition must be lossless: it must be possible to reconstruct the original relation from the relations in the decomposition
  - We will see why
Decomposition

• Schema $\mathbf{R} = (\mathbf{R}, \mathbf{F})$
  – $\mathbf{R}$ is set of attributes
  – $\mathbf{F}$ is set of functional dependencies over $\mathbf{R}$
    • Each key is described by a FD
• The decomposition of schema $\mathbf{R}$ is a collection of schemata $\mathbf{R}_i = (\mathbf{R}_i, \mathbf{F}_i)$ where
  – $\mathbf{R} = \bigcup \mathbf{R}_i$ for all $i$ (no new attributes)
  – $\mathbf{F}_i$ is a set of functional dependencies involving only attributes of $\mathbf{R}_i$
  – $\mathbf{F}$ entails $\mathbf{F}_i$ for all $i$ (no new FDs)
• The decomposition of an instance, $\mathbf{r}$, of $\mathbf{R}$ is a set of relations $\mathbf{r}_i = \mathbf{R}_i(\mathbf{r})$ for all $i$

Example Decomposition

Schema $(\mathbf{R}, \mathbf{F})$ where

$\mathbf{R} = \{\text{SSN, Name, Address, Hobby}\}$

$\mathbf{F} = \{\text{SSN} \to \text{Name, Address}\}$

can be decomposed into

$\mathbf{R}_1 = \{\text{SSN, Name, Address}\}$

$\mathbf{F}_1 = \{\text{SSN} \to \text{Name, Address}\}$

and

$\mathbf{R}_2 = \{\text{SSN, Hobby}\}$

$\mathbf{F}_2 = \{\}$

Lossless Schema Decomposition

• A decomposition should not lose information
• A decomposition $(\mathbf{R}_1, \ldots, \mathbf{R}_n)$ of a schema, $\mathbf{R}$, is lossless if every valid instance, $\mathbf{r}$, of $\mathbf{R}$ can be reconstructed from its components:

$$\mathbf{r} = \mathbf{r}_1 \times \mathbf{r}_2 \times \cdots \times \mathbf{r}_n$$

• where each $\mathbf{r}_i = \pi_{\mathbf{R}_i}(\mathbf{r})$

Lossy Decomposition

The following is always the case (Think why?):

$$\mathbf{r} \subseteq \mathbf{r}_1 \times \mathbf{r}_2 \times \cdots \times \mathbf{r}_n$$

But the following is not always true:

$$\mathbf{r} \supseteq \mathbf{r}_1 \times \mathbf{r}_2 \times \cdots \times \mathbf{r}_n$$

Example: $\mathbf{r} \supseteq \mathbf{r}_1 \times \mathbf{r}_2 \times \cdots \times \mathbf{r}_n$

$\begin{array}{c|c|c}
\text{SSN} & \text{Name} & \text{Address} \\
1111 & \text{Joe} & 1 \text{Pine} \\
2222 & \text{Alice} & 2 \text{Oak} \\
3333 & \text{Alice} & 3 \text{Pine} \\
\end{array}$

$\begin{array}{c|c|c}
\text{SSN} & \text{Name} & \text{Address} \\
1111 & \text{Joe} & 1 \text{Pine} \\
2222 & \text{Alice} & 2 \text{Oak} \\
3333 & \text{Alice} & 3 \text{Pine} \\
\end{array}$

The tuples $(2222, \text{Alice, 3 Pine})$ and $(3333, \text{Alice, 2 Oak})$ are in the join, but not in the original

Lossy Decompositions: What is Actually Lost?

• In the previous example, the tuples $(2222, \text{Alice, 3 Pine})$ and $(3333, \text{Alice, 2 Oak})$ were gained, not lost!
  – Why do we say that the decomposition was lossy?

• What was lost is information:
  – That 2222 lives at 2 Oak: In the decomposition, 2222 can live at either 2 Oak or 3 Pine
  – That 3333 lives at 3 Pine: In the decomposition, 3333 can live at either 2 Oak or 3 Pine

Testing for Losslessness

• A (binary) decomposition of $\mathbf{R} = (\mathbf{R}, \mathbf{F})$ into $\mathbf{R}_1 = (\mathbf{R}_1, \mathbf{F}_1)$ and $\mathbf{R}_2 = (\mathbf{R}_2, \mathbf{F}_2)$ is lossless if and only if:

  – either the FD
    • $(\mathbf{R}_1 \cap \mathbf{R}_2) \to \mathbf{R}_1$ is in $\mathbf{F}^+$
  – or the FD
    • $(\mathbf{R}_1 \cap \mathbf{R}_2) \to \mathbf{R}_2$ is in $\mathbf{F}^+$
Example

Schema \((R, F)\) where
\[ R = \{\text{SSN, Name, Address, Hobby}\} \]
\[ F = \{\text{SSN} \rightarrow \text{Name, Address}\} \]
can be decomposed into
\[ R_1 = \{\text{SSN, Name, Address}\} \]
\[ F_1 = \{\text{SSN} \rightarrow \text{Name, Address}\} \]

and
\[ R_2 = \{\text{SSN, Hobby}\} \]
\[ F_2 = \{\} \]
Since \(R_1 \cap R_2 = \text{SSN}\) and \(\text{SSN} \rightarrow R_1\) the decomposition is lossless

Intuition Behind the Test for Losslessness

\[ R_1 \cap R_2 = \{\text{SSN}\} \]
\[ R_1 \quad R_2 \]
\[ r_1 \quad r_2 \]

But \(\text{card}(r_1) \geq \text{card}(\text{projection})\) (since \(r_1\) is a projection of \(r\)) and therefore \(\text{card}(r) \geq \text{card}(r_1 \times r_2)\)

Hence \(r = r_1 \times r_2\)

Dependency Preservation

If \(f\) is an FD in \(F\), but \(f\) is not in \(F_1 \cup F_2\), there are two possibilities:

\(-f \notin (F_1 \cup F_2)^+\)

- If the constraints in \(F_1\) and \(F_2\) are maintained, \(f\) will be maintained automatically.

\(-f \notin (F_1 \cup F_2)^+\)

- \(f\) can be checked only by first taking the join of \(r_1\) and \(r_2\). This is costly.

Example

Schema \((R, F)\) where
\[ R = \{\text{SSN, Name, Address, Hobby}\} \]
\[ F = \{\text{SSN} \rightarrow \text{Name, Address}\} \]
can be decomposed into
\[ R_1 = \{\text{SSN, Name, Address}\} \]
\[ F_1 = \{\text{SSN} \rightarrow \text{Name, Address}\} \]

and
\[ R_2 = \{\text{SSN, Hobby}\} \]
\[ F_2 = \{\} \]
Since \(F = F_1 \cup F_2\), the decomposition is dependency preserving.
Example

- Schema: \((ABC; F)\), \(F = \{A \rightarrow C, B \rightarrow C, C \rightarrow B\}\)
- Decomposition:
  - \((AC, F_1), F_1 = \{A \rightarrow C\}\)
  - \((BC, F_2), F_2 = \{B \rightarrow C, C \rightarrow B\}\)
- \(A \rightarrow B \notin (F_1 \cup F_2)\), but \(A \rightarrow B \in (F_1 \cup F_2)^*\).
- \(F^* = (F_1 \cup F_2)^*\) and thus the decompositions is still dependency preserving.

BCNF Decomposition Algorithm

**Input:** \(R = \langle R; F \rangle\)

**Decomp := R**

**while** there is \(S = (S; F^*) \in \text{Decomp} \) and \(S\) not in BCNF **do**

1. Find \(X \rightarrow Y \notin F^*\) that violates BCNF  \(X\) isn’t a superkey in \(S\)
2. Replace \(S\) in \(\text{Decomp}\) with \(S_1 = (XY; F_1)\), \(S_2 = (S - (Y - X); F_2)\)
3. **end**

**return** \(\text{Decomp}\)

A Larger Example

Given: \(R = \langle R; F \rangle\) where \(R = ABCDEGHK\) and \(F = \{ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow K, K \rightarrow ADH, BH \rightarrow GE\}\)

**step 1:** Find a FD that violates BCNF
- Not \(ABH \rightarrow C\) since \(\{ABH\}^*\) includes all attributes \((BH\) is a key\)
- \(A \rightarrow DE\) violates BCNF since \(A\) is not a superkey \((A^* = ADE)\)

**step 2:** Split \(R\) into:
- \(R_1 = \langle ADE, F_1 = \{A \rightarrow DE\} \rangle\)
- \(R_2 = \langle ABCGHK, F_2 = \{ABH \rightarrow C, BGH \rightarrow K, K \rightarrow AH, BH \rightarrow G\} \rangle\)

Note 1: \(R_1\) is in BCNF.
Note 2: Decomposition is **lossless** since \(A\) is a key of \(R_1\).
Note 3: FDs \(K \rightarrow D\) and \(BH \rightarrow E\) are not in \(F_1\) or \(F_2\). But both can be derived from \(F_1 \cup F_2\)

- (E.g. \(K \rightarrow A\) and \(A \rightarrow D\) implies \(K \rightarrow D\))

Hence, decomposition is **dependency preserving.**

Example

- HasAccount \((\text{AcctNum}, \text{ClientId}, \text{OfficeId})\)
  - \(f_1: \text{AcctNum} \rightarrow \text{OfficeId}\)
  - \(f_2: \text{ClientId}, \text{OfficeId} \rightarrow \text{AcctNum}\)

**Decomposition:**
- \(R_1 = \langle \text{AcctNum, OfficeId}, \langle \text{AcctNum} \rightarrow \text{OfficeId} \rangle\rangle\)
- \(R_2 = \langle \text{AcctNum}, \text{ClientId} \rangle\)

**Decomposition is lossless:**
- \(R_1 \cap R_2 = \langle \text{AcctNum} \rangle\) and \(\text{AcctNum} \rightarrow \text{OfficeId}\)
- In BCNF
- Not dependency preserving: \(f_1 \notin (F_1 \cup F_2)^*\)
- HasAccount does not have BCNF decompositions that are both lossless and dependency preserving! (Check, eg. by enumeration)
- Hence: BCNF+lossless+dependency preserving decompositions are not always achievable!

Simple Example

- HasAccount:

  - \((\text{ClientId, OfficeId, AcctNum})\)
  - \((\text{ClientId, OfficeId} \rightarrow \text{AcctNum})\)
  - \((\text{AcctNum} \rightarrow \text{OfficeId})\)

**Decompose using** \(\text{AcctNum} \rightarrow \text{OfficeId}\):

- \((\text{OfficeId, AcctNum})\)
- BCNF: \(\text{AcctNum}\) is key
- FD: \(\text{AcctNum} \rightarrow \text{OfficeId}\)
- BCNF (only trivial FDs)

Example (con’t)

Given: \(R_3 = \langle ABCGHK, \{ABH \rightarrow C, BGH \rightarrow K, K \rightarrow AH, BH \rightarrow G\} \rangle\)

**step 1:** Find a FD that violates BCNF.
- Not \(ABH \rightarrow C\) or \(BGH \rightarrow K,\) since \(BH\) is a key of \(R_3\), \(K \rightarrow AH\) violates BCNF since \(K\) is not a superkey \((K^* = AH)\)

**step 2:** Split \(R_3\) into:
- \(R_{21} = \langle KAH, F_{21} = \{K \rightarrow AH\} \rangle\)
- \(R_{22} = \langle BCGK, F_{22} = \{\} \rangle\)

Note 1: Both \(R_{21}\) and \(R_{22}\) are in BCNF.
Note 2: The decomposition is **lossless** (since \(K\) is a key of \(R_{21}\))
Note 3: FDs \(ABH \rightarrow C, BGH \rightarrow K, BH \rightarrow G\) are not in \(F_{21}\) or \(F_{22}\) and they can’t be derived from \(F_{21} \cup F_{22}\).

Hence the decomposition is **not** dependency-preserving.
Properties of BCNF Decomposition Algorithm

Let \( X \rightarrow Y \) violate BCNF in \( R = (R, F) \) and \( R_1 = (R_1, F_1), R_2 = (R_2, F_2) \) is the resulting decomposition. Then:

- There are fewer violations of BCNF in \( R_1 \) and \( R_2 \) than there were in \( R \)
  - \( X \rightarrow Y \) implies \( X \) is a key of \( R_1 \)
  - Hence \( X \rightarrow Y \in F_1 \) does not violate BCNF in \( R_1 \), and, since \( X \rightarrow Y \notin F_2 \) does not violate BCNF in \( R_2 \), either
  - Suppose \( f \in X' \rightarrow Y' \) and \( f \notin F \) doesn’t violate BCNF in \( R \).
  - If \( f \in F_1 \) or \( F_2 \), it does not violate BCNF in \( R_1 \) or \( R_2 \), either since \( X' \) is a superkey of \( R \) and hence also of \( R_1 \) and \( R_2 \).

A BCNF decomposition is not necessarily dependency preserving
But always lossless:

- \( R_1 \cap R_2 = X \), \( X \rightarrow Y \), and \( R_1 = XY \)

- BCNF+lossless+dependency preserving is sometimes unachievable (recall HasAccount)

Minimal Cover

- A minimal cover of a set of dependencies, \( F \), is a set of dependencies, \( U \), such that:
  - \( U \) is equivalent to \( F \) (\( F^+ = U^+ \))
  - All FDs in \( U \) have the form \( X \rightarrow A \) where \( A \) is a single attribute
  - It is not possible to make \( U \) smaller (while preserving equivalence) by
    - Deleting an FD
    - Deleting an attribute from an FD (either from LHS or RHS)

- FDs and attributes that can be deleted in this way are called redundant

Computing Minimal Cover

- Example: \( F = \{ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, BGH \rightarrow L, L \rightarrow AD, E \rightarrow L, BH \rightarrow E\} \)

  - step 1: Make RHS of each FD into a single attribute
    - Algorithm: Use the decomposition inference rule for FDs
    - Example: \( L \rightarrow AD \) replaced by \( L \rightarrow A, L \rightarrow D \); \( ABH \rightarrow CK \) by \( ABH \rightarrow C, ABH \rightarrow K \)

  - step 2: Eliminate redundant attributes from LHS.
    - Algorithm: If FD \( XB \rightarrow A \in F \) (where \( B \) is a single attribute) and \( X \rightarrow A \) is entailed by \( F \), then \( B \) was unnecessary
    - Example: Can an attribute be deleted from \( ABH \rightarrow C \) ?
      - Compute \( ABH + BH \rightarrow C \)
      - Since \( C \in (BH)^+ \), \( BH \rightarrow C \) is entailed by \( F \) and \( A \) is redundant in \( ABH \rightarrow C \)

  - step 3: Delete redundant FDs from \( F \)
    - Algorithm: If \( F - \{f\} \) entails \( f \), then \( f \) is redundant
      - If \( f \) is \( X \rightarrow A \) then check if \( A \in X^+ \)
      - Example: \( BGH \rightarrow L \) is entailed by \( E \rightarrow L, BH \rightarrow E \), so it is redundant

- Note: The order of steps 2 and 3 cannot be interchanged!! See the textbook for a counterexample

Computing Minimal Cover (con’t)
Synthesizing a 3NF Schema

Starting with a schema \( R = (R, F) \)

- **step 1**: Compute a minimal cover, \( U \), of \( F \). The decomposition is based on \( U \), but since \( U^+ = F^+ \) the same functional dependencies will hold
  - A minimal cover for \( F = \{ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, BGH \rightarrow L, \, L \rightarrow AD, \, E \rightarrow L, BH \rightarrow E\} \)
    is \( U = \{BH \rightarrow C, BH \rightarrow K, \, A \rightarrow D, C \rightarrow E, \, L \rightarrow A, \, E \rightarrow L\} \)

- **step 2**: Partition \( U \) into sets \( U_1, U_2, \ldots, U_n \) such that the LHS of all elements of \( U_i \) are the same
  - \( U_1 = \{BH \rightarrow C, BH \rightarrow K\} \)
  - \( U_2 = \{A \rightarrow D\} \)
  - \( U_3 = \{C \rightarrow E\} \)
  - \( U_4 = \{L \rightarrow A\} \)
  - \( U_5 = \{E \rightarrow L\} \)

- **step 3**: For each \( U_i \) form schema \( R_i = (R_i, U_i) \), where \( R_i \) is the set of all attributes mentioned in \( U_i \)
  - Each FD of \( U_i \) will be in some \( R_i \). Hence the decomposition is dependency preserving
  - \( R_1 = (BHCK; \, BH \rightarrow C, BH \rightarrow K) \)
  - \( R_2 = (AD; \, A \rightarrow D) \)
  - \( R_3 = (CE; \, C \rightarrow E) \)
  - \( R_4 = (AL; \, L \rightarrow A) \)
  - \( R_5 = (EL; \, E \rightarrow L) \)

- **step 4**: If no \( R_i \) is a superkey of \( R \), add schema \( R_0 = (R_0, \{\}) \) where \( R_0 \) is a key of \( R \).
  - \( R_0 = (BH, \{\}) \)
    - \( R_0 \) might be needed when not all attributes are necessarily contained in \( R_1, R_2, \ldots, R_n \)
      - A missing attribute, \( A \), must be part of all keys (since it’s not in any FD of \( U \), deriving a key constraint from \( U \) involves the augmentation axiom)
  - \( R_0 \) might be needed even if all attributes are accounted for in \( R_1, R_2, \ldots, R_n \)
    - Example: \( (ABCD; \, A \rightarrow B, C \rightarrow D) \)
      - Step 3 decomposition: \( R_1 = (AB; \, A \rightarrow B) \)
      - Lossy! Need to add \( \{\} \), for lossless

- **step 4** guarantees lossless decomposition.

BCNF Design Strategy

- The resulting decomposition, \( R_1, R_2, \ldots, R_n \), is
  - Dependency preserving (since every FD in \( U \) is a FD of some schema)
  - Lossless (although this is not obvious)
  - In 3NF (although this is not obvious)
- Strategy for decomposing a relation
  - Use 3NF decomposition first to get lossless, dependency preserving decomposition
  - If any resulting schema is not in BCNF, split it using the BCNF algorithm (but this may yield a non-dependency preserving result)

Normalization Drawbacks

- By limiting redundancy, normalization helps maintain consistency and saves space
- But performance of querying can suffer because related information that was stored in a single relation is now distributed among several
- **Example**: A join is required to get the names and grades of all students taking CS305 in S2002.

```sql
SELECT S.Name, T.Grade
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND
T.CrsCode = 'CS305' AND T.Semester = 'S2002'
```
Denormalization

- **Tradeoff**: Judiciously introduce redundancy to improve performance of certain queries
- **Example**: Add attribute `Name` to `Transcript`

```sql
SELECT T.Name, T.Grade
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Semester = 'S2002'
```

- Join is avoided
- If queries are asked more frequently than `Transcript` is modified, added redundancy might improve average performance
- But, `Transcript` is no longer in BCNF since key is `(StudId, CrsCode, Semester)` and `StudId`, `Name`, `T.CrsCode` = `CS305`, `T.Semester` = `S2002`

Multi-Valued Dependency

- **Problem**: multi-valued (or binary join) dependency
- **Definition**: If every instance of schema `R` can be (losslessly) decomposed using attribute sets `(X, Y)` such that:
  - `r = \pi_X(r) \bowtie \pi_Y(r)`
  - then a multi-valued dependency
  - `R = \pi_X(R) \bowtie \pi_Y(R)` holds in `r`

Ex: `Person = SSN,PhoneN(Person) \bowtie SSN,ChildSSN(Person)`

Fourth Normal Form (4NF)

- A schema is in **fourth normal form** (4NF) if for every multi-valued dependency
  - `R = X \bowtie Y`
  - it is in BCNF (since there are no non-trivial FDs)
  - Redundancy is due to set valued attributes (in the E-R sense), not because of the FDs

4NF Implies BCNF

- **Intuition**: if `X \bowtie Y \rightarrow R`, there is a unique row in relation `r` for each value of `X \bowtie Y` (hence no redundancy)
  - Ex: `SSN` does not uniquely determine `PhoneN` or `ChildSSN`, thus `Person` is not in 4NF.
- **Solution**: Decompose `R` into `X` and `Y`
  - Decomposition is lossless – but not necessarily dependency preserving (since 4NF implies BCNF – next)